Unifying Textual and Visual: a Theoretical Account of the Visual Perception of Programming Languages

Stéphane Conversy
Université de Toulouse - ENAC, France
stephane.conversy@enac.fr

Abstract

Firm principles which can be relied on to analyze and discuss textual and graphical code representations are still missing. We propose a framework relying on ScanVis, an extension of the Semiology of Graphics that models the perception and scanning of abstract graphics, to model and to provide plausible explanations of phenomena pertaining to the visual perception of representations of code. This framework unifies many aspects of the visual layout and appearance of programming languages and reveals similarities and substantial differences in the visual operations required by those notations. We also show how the framework may help compare and generate representations of programming languages with respect to visual perception. This work suggests that the gap between textual and graphical languages is narrow, and that all kind of programming languages should rely on the capability of the human visual system.

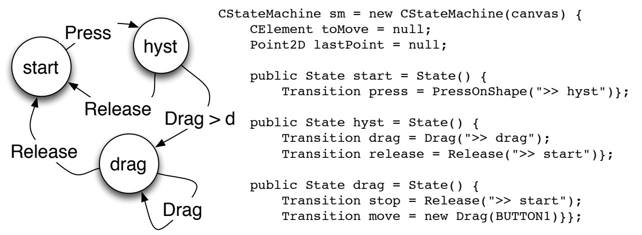
Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Programming Languages; Visual Perception

1. Introduction

An implicit but important aspect of programming languages is that they must support the production of readable programs [1]: "Programs must be written for people to read, and only incidentally for machines to execute [2]". Programmers read a program by looking at its 'code', i.e., the representation of the program on the screen, perceptible by their eyes. Both textual and so-called 'visual' representations of programs on the screen employ various graphical 'features': texts, shapes, alignments, colors, arrows, etc. (fig. 1). Those

Onward! 2014, October 20–24, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-3210-1/14/10...$15.00.
http://dx.doi.org/10.1145/10.1145/2661136.2661138



```
CStateMachine sm = new CStateMachine(canvas) {
    CElement toMove = null;
    Point2D lastPoint = null;

    public State start = State() {
        Transition press = PressOnShape(">> hyst")};

    public State hyst = State() {
        Transition drag = Drag(">> drag");
        Transition release = Release(">> start")};

    public State drag = State() {
        Transition stop = Release(">> start");
        Transition move = new Drag(BUTTON1)}};
```

Figure 1. Two representations of the same program.

visual features are often considered 'aesthetic sugar' that do not map semantics (e.g., a colored textual C program), but they can also be part of the syntax (e.g. indented Python code, arrows in state machines, colored Petri-nets).

As with any visual scene, the performance of programmers reading textual or visual programs depends on their performance in perceiving the visual features presented on the screen. However, few works exist that help analyze those features and their impact on performance (an exception is [3]). Instead, programming specialists mention 'aesthetics' or 'personal preferences' [1, 4] and warn about the possible 'danger of religious wars' when dealing with the topic [9]. The use of such terms signals a possible lack of foundation for addressing the phenomenon of code perception and how this may help or hinder programmers' performance.

This paper investigates the principles of programming languages that underpin the practice of code representation: we aim to find the science in the art, rather than finding the art in the science as advocated in [4]. We show how ScanVis [5], an extension of the Semiology of Graphics [6] that models the perception and scanning of abstract graphics, may help model, compare and generate visual representations of programming languages with respect to human perception. The expected benefits of this work from a scientific point of view are a better understanding of the phenomenon of code perception, the unification of the concepts used in the literature, and accurate definitions of these concepts. The outcome for end programmers would be better designed programming languages and IDEs with respect to this concern.

We focus on the representation of 'a single page' of code, though current trends in the visualization of code focus on the management and representation of large-scale programs and despite the fact that we appreciate that interaction with code is important [5, 7]. However, we think that representation at the level of the page is overlooked: understanding a single page of code is still required since the very act of programming (i.e., editing code) is often done at this level.

2. Related Work

Reading code is a complex process that involves many aspects. We have selected a number of works that address formatting, performance in reading, differences between textual and visual languages, and frameworks to analyze them.

2.1 Formatting and pretty-printing

'Formatting well' is often advised and discussed in early fundamental papers about programming languages (e.g., the discussion in [8]): "Code formatting is about communication, and communication is the professional developer's first order of business" [9]. In a recent work, formatting is still referred to as an 'art' [4]. Actually, the problem of program representation goes well beyond code formatting and refers to the more general problem of the visual perception of the code by the programmer.

2.2 Performance in reading programming languages

A number of visual designs have been proposed to improve reading performance [10–13]. Indentation length has been experimentally shown to have an impact on the comprehension of code: 2- and 4-space indentation enables reader to better understand the code than 6-space indentation, for both novice and expert readers [14]. Eye tracking has been used to observe and measure programmers switching between a view of the code and a view presenting an animated algorithm [15], and to determine whether identifier-naming conventions (i.e., camelCase and under_score) affect code comprehension [16]. Background colors may improve comprehension of preprocessors' #ifdef directives [17].

Moher et al. observed that "performance was strongly dependent on the layout of the Petri nets. In general, the results indicate that the efficiency of a graphical program representation is not only task-specific, but also highly sensitive to seemingly ancillary issues such as layout and the degree of factoring" [18]. Green et al. found that textual representations outperformed LabView for each and every subject [19]. Their explanation is that "the structure of the graphics in the visual programs is, 'paradoxically', harder to scan than in the text version". LabView and its G language have been studied "in the wild" [20]. Respondents declared that G is easier to read than textual programming languages since it provides an overview (a gestalt view) and clarifies structure. However, respondents also say that it is very easy to create messy, cluttered, hard to read spaghetti code and that sequence structures tend to be cryptic or obscure.

2.3 Differences between textual and visual languages

Researchers have already wondered where the actual differences between textual and visual languages lie. In [21] Petre argues that the differences in effectiveness between textual and visual languages "lie not so much in the textual-visual distinction as in the degree to which specific representations support the conventions experts expect." As Petre observed, programmers can find gestalt patterns in textual representations [21]. Much of "what contributes to comprehensibility of a graphical representation is not part of the formal notation but a 'secondary notation': layout, typographical cues and graphical enhancements." Petre adds that "the secondary notations (e.g., layout) are subject to individual skills (i.e., learned ones) and make the difference between novices and experts. What is required in addition is good use of secondary notation, which like 'good design' is subject to personal style and individual skill" [21]. We take an alternative point of view: we argue here that even if skills can be learned, the basic visual capability is enough to explain some of the ease or difficulty programmers experience in deciphering a program representation.

2.4 Analysis frameworks

There have been several attempts at building metrics for software readability. In the metric from [22], a few features can be considered perceptual (commas, spaces, indentation), but most are based on the semantics of the code. The cognitive dimensions of notation (CDN) is a framework that helps designers analyze interactive tools, including programming environments and languages [7]. CDN targets cognitive and interactive aspects as opposed to perceptual aspects: the graphic and perceptual concerns are addressed partly in the "secondary notation" and "visibility" dimensions.

Gestalt is a well-known framework that explains the phenomena underlying pattern perception. Gestalt may be used to explain how programmers perceive patterns in their code, but we found that Gestalt could not report about all perception phenomena. So-called pre-attentive features also have a role in the perception of code [23]. The Physics of Notations framework focuses on the properties of notations [3]. It addresses numerous aspects of graphical properties and can be considered as a broader framework than the one presented here. One of those aspects of the Physics of Notations relies on the Semiology of Graphics. We discuss the relationship between Semiology of Graphics, ScanVis, pre-attentiveness and the Physics of Notations in the following section.

3. Framework

The framework we used relies on the Semiology of Graphics and ScanVis.

3.1 Semiology of Graphics

The Semiology of Graphics (SoG) is a theory of abstract drawings (i.e., drawings that do not imitate a natural scene) such as maps and bar charts [6]. A part of this theory explains the perceptual phenomena and properties underlying the act of visualizing 2D abstract drawings. SoG relies on the characterization of data to be represented (the data type: nominal, ordered, and quantitative), and the perceptual properties used in a drawing, such as color or shape.

SoG first defines a set of concepts and a vocabulary. Drawings are a set of 2D *marks* (points, lines or zones) lying over a background. Marks vary according to *visual variables* such as *position (Xpos and Ypos), shape, color, luminosity, size, orientation [6], enclosures and lines that link two marks [24]*. Visual variables are characterized by their *perceptual properties*, and can be: *selective* – enable a viewer to assimilate and differentiate marks instantaneously (e.g., all red marks) (fig. 2); *ordered* – enables a viewer to rank marks perceptually (e.g., from light to dark) (fig. 3); and *quantitative* – enable a viewer to quantify differences between marks perceptually (e.g. twice as large) (fig. 3).

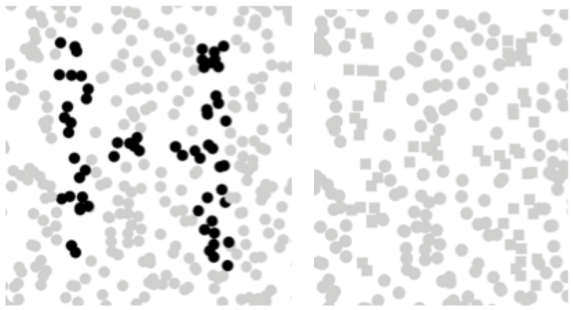


Figure 2. By default marks are circular and light. **Left:** Some marks are dark to produce an H. Luminosity is selective: the H letter emerges because the eye discriminates two groups of marks (light and dark) instantaneously. **Right:** Marks at the same locations, forming the H, are square. Shape is not selective so the H letter does not emerge.

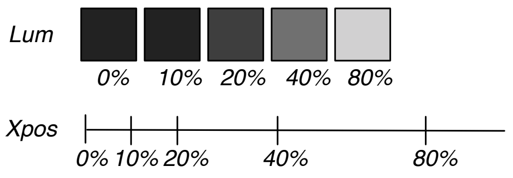


Figure 3. Top: Luminosity is ordered but cannot be perceived quantitatively. **Bottom:** Position is quantitative: one can perceive the ratio and the difference between various X positions (pos. 80% is 2x pos. 40% and 4x pos. 20%).

SoG also edicts a number of rules. All visual variables except shape and link may be selective. Fig. 2 illustrates selectivity: a H form emerges from marks varying in luminosity (left) but not from the same marks varying in shape (left). All visual variables except shape, link and color may be ordered (colors are ordered in a limited spectrum only). Xpos, Ypos, angle, length, size may be quantitative to various degrees, as demonstrated experimentally [25]. Fig. 4 summarizes the properties of visual variables. However, as we will see, any statement about the properties of a particular (set of) mark(s) should be made while taking into account the relations between all marks. The performance of readers at selecting, ordering or quantifying depends on the number of values that can be differentiated (e.g., 5 levels of luminosity for selection, 20 levels of luminosity for order), the difference between each value (larger differences produce better performance), and the spatial distance between marks (smaller distances produce better performance).

	shape	lum	color	pos	size	link	cont
selective		✓	✓	✓	✓		✓
ordered		✓		✓	✓		✓
quantitative				✓	✓		

Figure 4. Summary of the properties of the visual variables

3.2 ScanVis

SoG may help design a representation that enables users to perceive multiple information elements at a single glance. Nevertheless, however well designed a representation is, it cannot be absolutely efficient: a representation may be well suited to a particular task, but may not be suitable for all tasks a user's activity requires. For such tasks, instead of perceiving the representation at one glance, the user falls back to scanning the representation to discover information.

ScanVis is a descriptive model of this kind of representation scanning [5]. It enables a designer to analyze and assess representation effectiveness with respect to a task. ScanVis relies on the decomposition of representation scanning into elementary visual operations: *enter into the representation* by transforming the conceptual task at hand ("how long will I wait?") into a reading task ("find the time corresponding to my bus route"), *seek a subset of marks* ("find the marks corresponding to my bus routes?"), *seek and navigate among a subset of marks* ("navigate into a row of text representing time in a time-table"), *unpack a mark and verify a predicate* ("what datum does this position reflect, and does it answer my question?"), *exit from the representation* ("this bus passes at this time, I need to compute the waiting time"), and *memorize* information ("I have to wait 2 minutes for this bus; remember this to compare with another bus"). Those operations constitute the vocabulary of ScanVis. Fig. 5 summarizes the operations.

Given a task and a representation, a designer can express the required sequence of visual operations to accomplish it. For example, fig. 6 shows two different representations of a bus schedule. Depending on the representation, the amount and the nature of visual scanning operations will differ (we do not discuss the differences between those particular two schedule representations here). The overlaid arrows and circles depict the visual scanning required on each to answer the same question: "how long will I wait for the next bus?". A small blue circle with a letter M depicts a need to memorize a datum e.g. the current time and the compatible

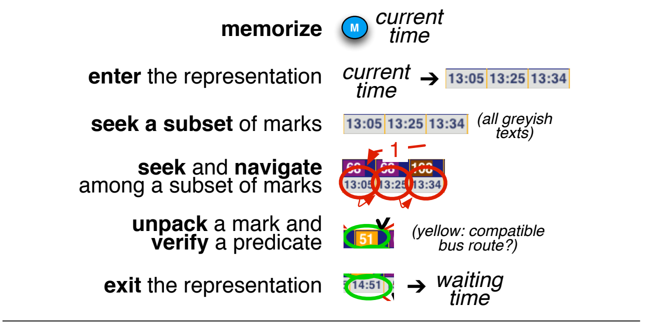


Figure 5. Summary of ScanVis operations

bus route numbers. Red [resp. green]-stroked circles refer to *predicates* that proved false [resp. true]. For instance, the ordered view requires the user to *memorize* the current time and the number and color of bus routes, to *navigate* (since they do not use a selective variable) the ordered list of times from left to right until the first time which is superior to the current time is found (*verify a predicate*), to *seek and navigate* among the colored cells from left to right until a compatible bus is found, to read (or *unpack*) the corresponding time, and to *exit the representation* by subtracting the current time to the found time and obtain the waiting time.

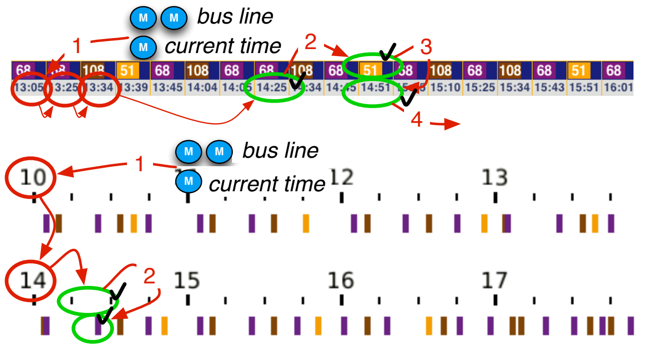


Figure 6. Two representations and the visual operations for the task "find how long I have to wait for the next bus."

ScanVis operations may be facilitated by the use of adequate visual properties as described by SoG e.g. *selective* visual variables to support *seeking and navigating among a subset of marks*. For example, if one wants to find out the next bus #51 in the representation at the bottom of fig. 6, one can visually *select* a subset of marks that are yellow (a selective variable) and that lie in the area whose position (a selective variable) corresponds roughly to the current time. One can then scan through the marks of this subset and hop from mark to mark until the next bus is found.

3.3 Comparison with other frameworks

The 'selective' property is equivalent to 'pre-attentive' (implicitly 'pre-attentive selection') or the 'pop-out' effect, but SoG is more comprehensive: preattentiveness not only concerns selection, but also ordering and quantification. As we will see, we also explicitly identified the data that the graphics represent (e.g. expression nesting, order of instructions, importance of text (comment/code)), which shows that the representations are indeed semiotic and not only preattentive. Compared to the Physics of Notations, our work offers a finer and more complete account of how SoG applies to graphical *and* textual code representation. Nevertheless, SoG alone cannot be used to assess code representation: the use of ScanVis and its emphasis on image scanning and precise task elicitation (e.g. "match parenthesis" vs "figure out the hierarchy of expressions", or "what is the next bus?" vs "how long will I wait?") makes the analysis finer.

4. Contribution

ScanVis & SoG (together referred to as 'the framework') have been applied to charts or visualizations of information in the past. The remainder of this paper is devoted to the application of the framework to programming languages.

Our goal is the production of knowledge about the phenomenon of code perception in order to ultimately design better programming languages and IDEs. Our contribution is threefold: (1) the demonstration that ScanVis & SoG can be applied to Programming Languages (2) the modeling of some phenomena pertaining to the properties attributed to code representation with ScanVis & SoG concepts (i.e, the translation of phenomena into ScanVis & SoG concepts and vocabulary) (3) the formulation of hypothetical explanations to these phenomena. The second and third contributions constitute the crux of the paper: the difficulty lies in *finding* and *formulating* a correct modeling in order to provide *detailed* and *plausible* hypothetical explanations of the phenomena. Though the explanations are hypothetical, they are the result of applying an existing framework that has been used successfully in other contexts.

As illustrated in the ScanVis section, assessing a particular visual representation of code requires identifying the set of reading tasks performed by the programmer. In the following sections, we discuss a number of visual representations together with code reading tasks and sub-tasks: visually structuring the code (sub-tasks: assimilate expression boundaries, figure out the hierarchy of expressions), understanding control flow (sub-tasks: given an instruction find next instruction, given a state find the out/in transition). We have devised the tasks with reference to some assumed assets of each representation. We believe that the tasks are appropriate but we do not claim completeness or perfect validity. In order to help the reader assess the significance of the proposed framework, we first describe its modeling power (what are the phenomena that the framework may capture?). We then describe its comparative power (how may it help assess or compare particular code representations?) and its generative power (how may it help explore the design space of code representation?) to illustrate possible uses.

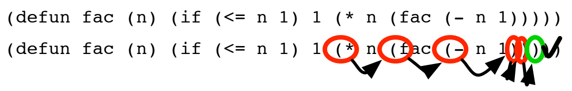
5. Modeling Code Perception

The goal of this section is to show how a number of phenomena may be appropriately captured with the concepts and vocabulary (in italics) of the framework. For this work, we used an abductive mode of inference (as opposed to an inductive or deductive mode). Abductions propose plausible explanations to observations [26, 27]. A presentation of abductions selects both the set of observations and the set of explanations to demonstrate correctness and completeness: "one can test the theory against as many specific situations and examples as possible, *looking for adequacy of explanation*. It is best if these examples are generated from *external* sources, as of course you tend to think of those that are already covered by your own theories" [28] (emphasis is ours).

We present the phenomena with popular and sometimes ambiguous *external* [28] sayings about code. By nature some of those sayings do not have a precise origin, but one can find them in scientific (referenced in the bibliography) and non-scientific texts (in footnotes). We chose these sayings and examples according to their variety and their ability to convey the explanations we devised. Though we do not claim that they are exhaustive, we argue that the number of examples suggests that the scope of the framework is significative, and that the difference between them suggests that the framework unifies a broad variety of representations. We then translate (i.e. model) those observations into *adequate* [28] concepts of the framework. If correct and comprehensive, a translation of ambiguous sayings into precise concepts is the first step toward the validation of the significance of the framework. Such a translation allows us to discuss in a precise manner the degree of truth of the sayings and the relationships between them.

5.1 Visually structuring the code

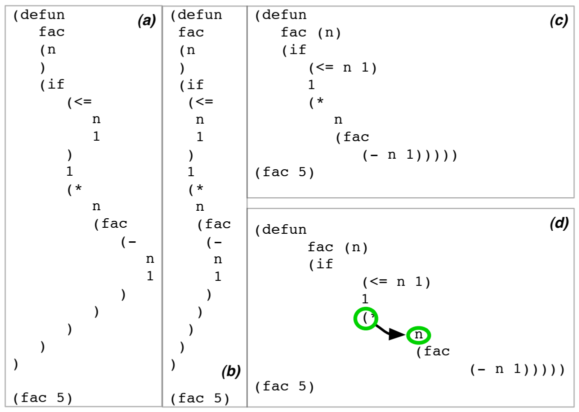
• *"Lots of Irritating Superfluous Parenthesis[1]"* LISP uses parentheses to structure code. Lists are designated with spaced expressions surrounded by opening and closing parentheses. Function composition uses compound parenthesizing.



```
(defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))
(defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))
```

Figure 7. Top: Parentheses as expression delimiters **Bottom:** ScanVis operations required to match an opening parenthesis.

LISP is reputedly difficult to read ([29] p65). The difficulty comes partly from the fact that list boundaries are depicted with two *shapes* (opening and closing parentheses, see fig. 7, top) which prevents fulfilling the task "figure out the [lisp] expressions" efficiently. This may be explained by the *selectivity* concept: since shape is *non-selective* within more than two variants, the use of parentheses prevents the perception of expression boundaries among all other letters and signs at a single glance and forces the programmer to *scan* the code linearly to discover them (fig. 7, bottom). Furthermore, the user is required to maintain a count of opening and closing parentheses during the scanning operation and *verify the predicate* "this symbol is a closing parenthesis and count is 0".



(a)

```
(defun
   fac
   (n
   )
   (if
      (<=
         n
         1
      )
      1
      (*
         n
         (fac
            (-
               n
               1
            )
         )
      )
   )
)

(fac 5)
```

(b)

```
(defun
 fac
 (n
 )
 (if
  (<=
   n
   1
  )
  1
  (*
   n
   (fac
    (-
     n
     1
    )
   )
  )
 )
)

(fac 5)
```

(c)

```
(defun
   fac (n)
   (if
      (<= n 1)
      1
      (*
         n
         (fac
            (- n 1)))))
(fac 5)
```

(d)

```
(defun
      fac (n)
      (if
            (<= n 1)
            1
            (*
               n
               (fac
                     (- n 1)))))
(fac 5)
```

Figure 8. Delimiters varying in *Xpos* and *Ypos*, which are both *selective visual variables*.(b) A smaller difference between Xpos values hinders *selection*. (c) Improving Xpos *selectiveness* by shortening spatial distances in *Ypos* or (d) with a larger indentation, at the expense of visual scanning, depicted with circles and arrows.

• *"Indentation makes structure obvious [14]"* In fig. 8 (a) the level of nesting depth is mapped to the *Xpos visual variable*. Matching parentheses are "vertically aligned", which is another way of expressing an *assimilation of Xpos values*. Since Xpos is *selective* in this case, the perception of expression boundaries is better than when using a shape. Selectivity depends on the *amount of difference* between values: shrinking the size of the indentation lowers the selective ability of Xpos (b). Reserving a line for a closing parenthesis alone lengthens the Ypos *spatial distance* with the matching opening parenthesis and weakens the selective property of the Xpos visual variable (i.e., it is difficult to perceive vertical alignment) (a and b); ignoring the parenthesis matching problem altogether shortens the distances and improves Xpos selectivity (c) [14]; more indentation improves selective perception of Xpos (d). However, the assumed improvement is supposedly accomplished at the expense of longer *scanning* from the beginning of a block to its first instruction, as experimentally assessed in [14]. Note that since Xpos is an ordered visual variable, such a representation may also facilitate the task "figure out the hierarchy of expressions".

• *"LabView's G language is intuitive[2]"* G combines large boxes that *enclose* other objects to specify a hierarchical structure (fig. 9), and *links* that connect the components in-

[1] en.wikipedia.org/wiki/Lisp_(programming_language)

[2] Editor's white paper http://www.ni.com/white-paper/14556/en

side and outside boxes (see [30] for more details). Enclosures may be "intuitive", but a more appropriate qualification is that they are *selective* in this case: one may grasp in a single glance which elements are part of a parent. Enclosure is also *ordered* and may help perception of a containment hierarchy.

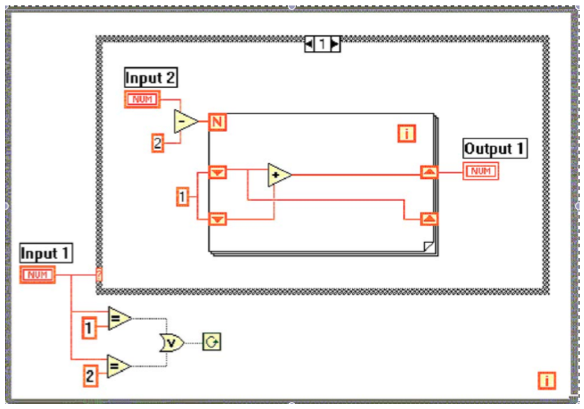
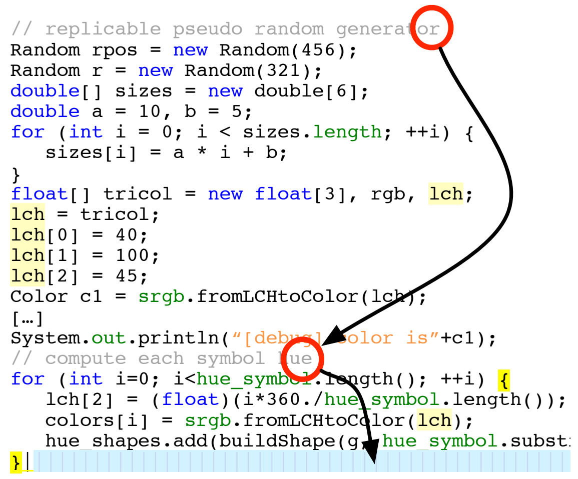


Figure 9. G language from LabView.

• *"Syntax highlighting improves readability*[3]*"* Fig. 10 shows a 'syntax-colored' textual representation of Java code in the NetBeans editor. Blue glyphs correspond to reserved keywords of the Java language and gray ones to comments. A yellow background corresponds to a variable to which the mouse pointer points.



```
// replicable pseudo random generator
Random rpos = new Random(456);
Random r = new Random(321);
double[] sizes = new double[6];
double a = 10, b = 5;
for (int i = 0; i < sizes.length; ++i) {
   sizes[i] = a * i + b;
}
float[] tricol = new float[3], rgb, lch;
lch = tricol;
lch[0] = 40;
lch[1] = 100;
lch[2] = 45;
Color c1 = srgb.fromLCHtoColor(lch);
[…]
System.out.println("[debug] color is"+c1);
// compute each symbol hue
for (int i=0; i<hue_symbol.length(); ++i) {
   lch[2] = (float)(i*360./hue_symbol.length());
   colors[i] = srgb.fromLCHtoColor(lch);
   hue_shapes.add(buildShape(g, hue_symbol.subst
}|
```

Figure 10. Colored editor and ScanVis for task "find a particular section of the code by reading comments"

Coloring all appearances of a variable the mouse is pointing at does make sense from a programming task point of view: this enables the programmer to efficiently identify all occurrences of this variable thanks to *selectivity* which in turn facilitates *seeking and navigation among a subset of marks* (here the subset of yellow marks). Similarly, adding a colored background to a brace enables the programmer to quickly locate where the corresponding brace is and assess the scope of a block. The gray color is lighter than the other ones: since luminosity is *selective*, this enables the user to rapidly *assimilate* and *differentiate* code from comments, and rapidly *seek and navigate among* grey marks e.g. to fulfill the task "find a particular section of the code by reading comments". This removes the need to *scan* the beginning of a line to check whether it begins with two slashes, a much more demanding visual operation since shape ('//') is not *selective*. In addition, the *order* of luminosity indicates an order of importance between code, comments, and background.

5.2 Understanding control flow

• *"Arrows make the instruction flow explicit [31]"* Fig. 1-left, shows a circle-and-arrow description of a Drag'n'Drop interaction with hysteresis [32]. There are three states ('start', 'hyst' and 'drag'), one transition from the state 'start', and two each from states 'hyst' and 'drag'. The instruction flow is depicted with marks: arrows. To fulfill the task "find the next instruction", a reader must *seek a subset of marks* (*links*) and *navigate* visually by following arrows. Similarly, Code Bubbles is an IDE that presents code with function snippets inside individual windows resembling 'bubbles' [33]. Users can juxtapose bubbles that contain related code. One use is to display the code of a callee in a bubble to the right of a bubble containing the caller, and display a link between the callee and the caller (fig. 11) to help find the next instruction after a call.

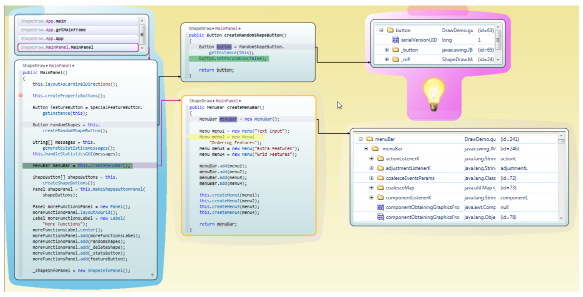


Figure 11. Code Bubble

• *"The control flow in C is implicit"* In a block of C instructions, a sequence of texts separated by semicolons denotes a sequence of instructions (fig. 10 for a C-like example). Often, programmers organize instructions one per line i.e., the Ypos visual variable maps the ordered sequence of the program counter: this helps the programmer visualize the evolution of the program counter path by *scanning* the textual instructions vertically. Thus, the task "given a particular instruction, what is the next instruction to be executed?" is efficiently supported by the representation since it uses a *selective*, *ordered* variable.

As mentioned above, arrows are supposed to make the instruction flow explicit [31], as opposed to the assumed implicitness of the flow of instructions of C. The framework

[3] en.wikipedia.org/wiki/Syntax_highlighting

enables us to be more precise: arrows are an explicit representation of the sequence *direction*, which is implicit in C with *Ypos*. However, links and arrows are no more explicit on the sequence of instructions than *Ypos*, since the *ordered visual variable Ypos* explicitly shows sequence already.

Indeed, a number of different representations can be unified with the same underlying principles. As seen above, instruction flow can be depicted using *Ypos* or links and arrows. Links and arrows are not *selective* visual variables: the reader is forced to follow the chain of links to understand the flow (fig. 12-a). The problem is similar with semicolons in C: as a *shape*, semicolons are not selective, and do not help discriminate between instructions. Links and arrows can be supplemented with so-called "alignment cues", e.g., using the *selective* property of *Ypos*. In this case, the visualization is equivalent to indented code in a C program (fig. 12-b). It is not necessary to show the arrows between successive instructions as this is redundant with the Ypos-aligned representation (fig. 12-c). However, keeping an arrow for the loop helps the reader *scan* up to the beginning of a loop, similar to box-and-arrow languages. Scratch [34] is a visual language with connectors on blocks which suggest how the latter should be put together (fig. 12-d). The connectors are similar to the arrows: they indicate the direction of the instruction sequence. The start of the loop can be perceived *selectively* with *color and containment*.

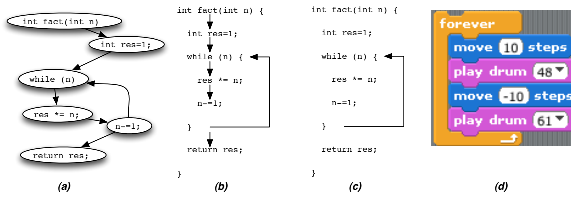


Figure 12. Arrows could have been used in C (b), as in box-and-arrow languages (a). Since arrows are redundant with the *ordered Ypos visual variable*, they can be removed, except for loops (c). Scratch uses similar visual variables (d).

• *"Spaghetti code [35]"* Reading code with links may be slow especially when arrows are numerous and entangled like spaghetti, which prevents arrows from being *selective*. In CodeBubble, hovering over a bubble highlights the connections and code lines that lead to it by changing the color or luminosity of the links. This turns a *non-selective* variable *selective* (from a link to a colored link) and helps readers comprehend the flow and navigate between instructions across functions.

• *"Befunge is an esoteric language*[4]*"* Befunge is a 2D textual language in which the flow is indicated by the four *shapes* <, >, ^, and v, which resemble arrows pointing in the four cardinal directions. Branching is specified by − (equivalent to < if the condition is true and to > otherwise) and — (equivalent to ^ if the condition is true and to v otherwise). However, not only is the flow not comprenhensible at once, but directional shapes are not *selective* and cannot be perceived instantaneously. As such, perhaps Befunge is not so esoteric since it may be considered as a missing link between textual and visual languages. This illustrates the unifying aspect of the framework.

[4] as qualified by its designer catseye.tc/node/Befunge-93.html

```
0&>:1-:v v *_$.@
  ^    _$>\:^
```

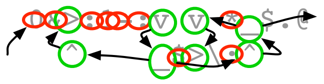


Figure 13. Factorial in befunge (l.); ScanVis operations (r.)

5.3 Understanding functionality

• *"Icons are easier to use than text [36]"* Icons are often considered easier to interpret than text [36]. Fig. 9 illustrates the use of an 'analog' [21] iconic vocabulary in LabView's control-flow structures. In this case, icons are differentiated with *shapes* (arrowheads, page corners, spirals, 'N' and 'I'). When icons vary according to *shape* only (a non-*selective* variable), one can only perform a slow *elementary reading* of a scene. In other visual languages, icons vary in shape but also along other *visual variables*, which may turn them selective (e.g. colored icons, or those that vary in luminosity like the asterisk-like shapes in fig. 14).

6. Application 1: Comparing Representations

The primary goal of the previous section was to show how various phenomena of code perception may be captured by the proposed framework concepts, and how the framework may provide plausible explanations to those phenomena. The present section is a first practical application of the modeling and shows how the framework may help compare code representations with respect to tasks by providing plausible explanations of their assets or weaknesses.

6.1 Visually structuring the code

This section reuses the LISP example in fig. 7, but the discussion also applies to any language that relies on character to delimit expressions or blocks such as C (parentheses or braces). Fig. 14 shows delimiters varying in *shape, hue, luminosity and hue*. As we have seen in the previous section, visually structuring the code may be related to two low-level tasks: "assimilate expression boundaries" and "figure out the hierarchy of expressions".

The second line of fig. 14 uses a unique boundary shape according to the level of depth of enclosure. The task "assimilate expression boundaries" may be easier to perform than with parentheses: since each level corresponds to a unique shape, there is no need to maintain a count anymore and it suffices to find the similar shape. However, finding a shape among other multiple different shapes may still be a difficult visual operation, since the *non-selectiveness* of symbols prevents matching at a single glance e.g. finding the diamond which closes the multiply expression is difficult and requires a careful horizontal scan. One may disagree and argue that we are able to see the right corresponding shape at once. Actually, shape can be *selective* but only when there are very few different marks (a dozen) with very few different shapes (two) [6]. If not convinced the reader is invited to refer to fig. 2. A longer, deeper nesting of expressions that requires more marks and more types of shape would exceed those numbers and inhibit *selectivity*.

```
1 (defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))
2 (defun fac ◁n▷ ◁if ◇<= n 1◇ 1 ◇* n ♡fac ☞- n 1☜♡◇▷)
3 (defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))
4 (defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))
5 (defun fac ┼n┼ ┼if ✳<= n 1✳ 1 ✳* n ✶fac ✷- n 1✷✶✳┼)
6 (defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))
```

Figure 14. Delimiters varying in shape (1:parentheses, 2:unique symbols), 3:hue, 4:lum, 5:shape+lum, 6:size.

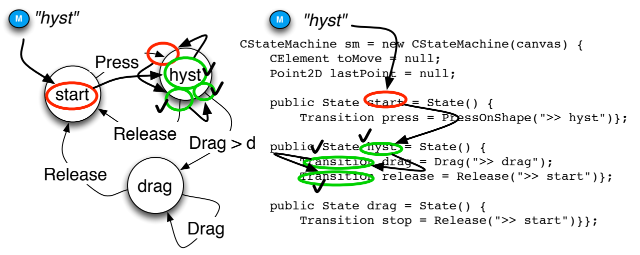
In fig. 14, the third line maps depth of nesting to unique *colors*. Since color is *selective* with few marks and few color values, this may enable the reader to *assimilate* at one glance all parentheses with the same level of depth. The fourth line maps *luminosity* to depth of nesting, while the sixth line maps the *size* of shapes to depth of nesting. In both cases, the *difference* of values (luminosity and size) may be too low to foster *selectiveness* and to help match parentheses.

However, it is questionable whether the task "assimilate expression boundaries" is worth facilitating: even if a reader correctly detects each opening and closing parenthesis, one must remember the discovered structure to make sense of it. If an appropriate visual variable was used instead to "figure out the hierarchy of expressions", the programmer could use that as an externalization of memory to recall the structure by accessing it immediately, in one glance. For instance, luminosity is *ordered* and may help perceive relative depth and thus help comprehend the hierarchy (line 4). The same statement holds for size, but not for shape (line 1 and 2) nor color (line 3) since shape is not *ordered* nor color in this case. The fifth line of fig. 14 uses a set of asterisk-like shapes with which *selection* and *ordering* seem to be effective: this is because they do not contain the same number of pixels and exhibit different levels of *luminosity*, a selective variable. This makes the delimiters *ordered and selective*.

6.2 Understanding control flow

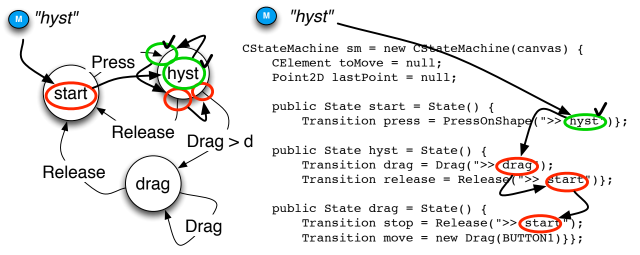
Code representations are used to fulfill multiple reading tasks. The following examples show how the framework may reveal differences when comparing two representations with respect to two different tasks.

Fig. 1-right shows the SwingState code describing the same Drag'n'drop interaction as in Fig. 1-left [32]. SwingStates is a textual language for describing state machines [37]. It relies on Java's anonymous class facility to be embedded seamlessly in regular Java code. The code is indented to facilitate perception of the states, the transitions from each state, and the clauses associated with the transitions. Fig. 15 compares the visual operations required for the task "what are the 'out' transitions for a particular state?" in the circles-and-arrows and SwingStates representations. In both, readers need to seek and navigate among states until they find the state of interest, then seek all transitions leaving this state. With circles-and-arrows, one may consider that large white circles are *selective* compared to other marks because of their *size and luminosity*. In the SwingStates code, indentation is also *selective*. Hence both representations help *seek a subset of marks*. Finding 'out' transition is more efficient in SwingStates code since all transitions are out transitions. With circle-and-arrows, one has to differentiate between links with and without arrowheads, laid around the circles. Links without arrowheads may be more difficult to differentiate than other marks.



```
CStateMachine sm = new CStateMachine(canvas) {
    CElement toMove = null;
    Point2D lastPoint = null;

    public State start = State() {
        Transition press = PressOnShape(">> hyst")};

    public State hyst = State() {
        Transition drag = Drag(">> drag");
        Transition release = Release(">> start")};

    public State drag = State() {
        Transition stop = Release(">> start")}};
```

Figure 15. ScanVis for task "find the 'out' transitions of state 'hyst'".



```
CStateMachine sm = new CStateMachine(canvas) {
    CElement toMove = null;
    Point2D lastPoint = null;

    public State start = State() {
        Transition press = PressOnShape(">> hyst")};

    public State hyst = State() {
        Transition drag = Drag(">> drag");
        Transition release = Release(">> start")};

    public State drag = State() {
        Transition stop = Release(">> start");
        Transition move = new Drag(BUTTON1)}};
```

Figure 16. ScanVis for task "find the 'in' transitions of state 'hyst'".

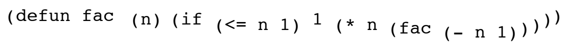
Fig. 16 illustrates the visual operations required for the task "what are the 'in' transitions for a particular state?" For the circle-and-arrows representation, the operations are almost identical to the operations required in the previous task. Finding the 'in' transition may be facilitated by the fact that arrowheads are *dark* and thus *selective*. With the SwingState code, the visual operations are very different: one has to find the name of the target states inside the transitions. Most of those names are on the right edge of the code, which helps seek and find them. Still, since they are textual, it may be difficult to navigate without risk of missing a name.

7. Application 2: Generating Representations

The present section is a second practical application of the modeling and shows how the framework may help generate representations with respect to tasks by providing plausible explanations of their assets or weaknesses. This section introduces a set of design principles that may be used to make new code representations emerge. We illustrate the design principles with a number of examples. We devised the design principles by providing plausible explanations on the assumed improvement.

Identify the task and apply selectivity only where needed. In the colored code fig. 10, using color for all keywords may not be related to any task the programmer needs to accomplish (e.g., find all 'for' loops). Of course, one may argue that the distinction helps assess that a keyword has been recognized as the programmer types it and that no lexical error has been made. However, fulfilling this task does not require a *selective* variable such as color. Instead, *elementary reading* with a non-selective variable such as a shape, or a typeface (e.g., 'unrecognized' in 'courier') is sufficient. This would reserve color, a scarce resource, for a more important use.

Try swapping visual variables. One way to generate representations is to explore the design space of code representation by swapping *visual variables* for unused ones. Fig. 17 illustrates an alternative representation with Ypos as visual variables instead of Xpos, color or luminosity. Since *Ypos is selective and ordered*, it may help the reader visualize the structure of the code, similarly to the more traditional use of the Xpos visual variable (indentation).

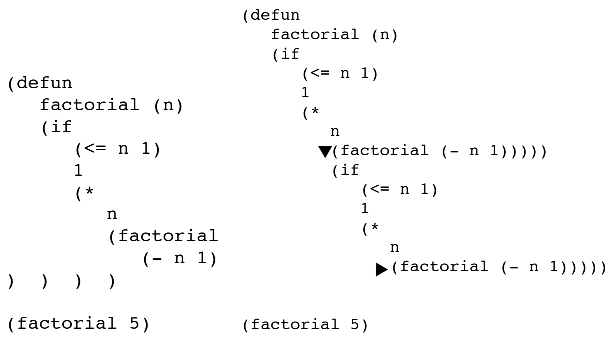


```
(defun fac (n) (if (<= n 1) 1 (* n (fac (- n 1)))))
```

Figure 17. Using Ypos as visual variables.

Shorten spatial distance. As mentioned previously, reducing spatial distance may improve selectivity. Fig. 8(c) shows a representation that shortens spatial distance, but does not support parenthesis matching, which may be annoying when trying to add a missing parenthesis. Fig. 18 is a representation that shortens the *spatial distance* while enabling easier parenthesis matching. Remarkably, while the parentheses match perceptually according to the Xpos visual variable, they do not match conceptually: for example, the opening parenthesis at the beginning of the 'defun' function conceptually matches the rightmost closing parenthesis on the penultimate line of the code, but is aligned with the closing parenthesis of the call to factorial. This illustrates that perception may prevail over the conceptual model, as long as semantics is preserved.

Another representation that reduces *distance* is shown in Fig. 18. With a debugger, the user can step inside the call of a function. This can be performed with a toggle arrow: when toggled, the code of the function unfolds under the call of the function. A similar feature could be used for static code; this would help understand how functions compose without the need to memorize the code surrounding the call of a function and to switch visually between the distant representations of the two functions.

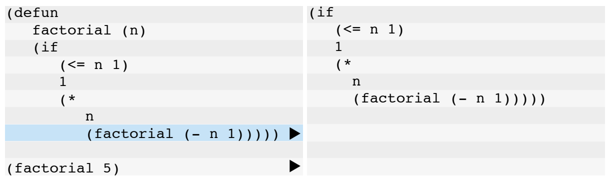


```
(defun
   factorial (n)
   (if
      (<= n 1)
      1
      (*
         n
         (factorial
            (- n 1)
)  )  )  )

(factorial 5)
```

```
(defun
   factorial (n)
   (if
      (<= n 1)
      1
      (*
         n
       ▼(factorial (- n 1)))))
         (if
            (<= n 1)
            1
            (*
               n
             ▶(factorial (- n 1)))))

(factorial 5)
```

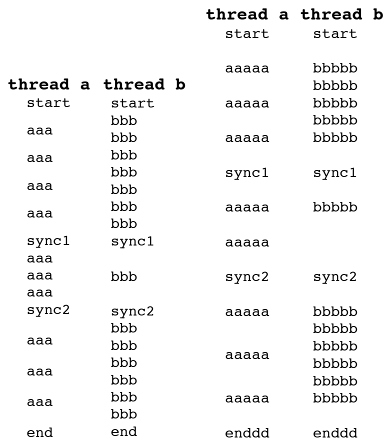
Figure 18. Shortening spatial distance: parentheses match perceptually, but do not match conceptually (left); 'Debugger view' of code (right).

One asset of such a 'tree-view' is that it shortens the *distance* between instructions before the call and instructions at the beginning of the function being called. However, it also expands the *distance* between the instructions before the call and after the call, especially when multiple functions are deployed. A 'browser' view ala SmallTalk can help show details and contexts of the call and shorten both distances (Fig. 19). Code Bubbles may be seen as a similar attempt to shorten the distance between calling and called code.



```
(defun                              (if
   factorial (n)                       (<= n 1)
   (if                                 1
      (<= n 1)                         (*
      1                                   n
      (*                                  (factorial (- n 1)))))
         n
         (factorial (- n 1))))) ▶

(factorial 5)                 ▶
```

Figure 19. Browser view of the 'factorial' function.



```
thread a  thread b
  start     start
            bbb
  aaa
            bbb
  aaa       bbb
            bbb
  aaa       bbb
            bbb
  aaa
            bbb
  sync1     sync1
  aaa
  aaa       bbb
  aaa
  sync2     sync2
            bbb
  aaa       bbb
            bbb
  aaa       bbb
            bbb
  aaa       bbb
  end       end
```

```
thread a  thread b
  start     start

  aaaaa     bbbbb
            bbbbb
  aaaaa     bbbbb
            bbbbb
  aaaaa     bbbbb

  sync1     sync1

  aaaaa     bbbbb

  aaaaa

  sync2     sync2

  aaaaa     bbbbb
            bbbbb
  aaaaa     bbbbb
            bbbbb
            bbbbb
  aaaaa     bbbbb

  enddd     enddd
```

Figure 20. Left: Ypos used as a selective variable: instructions are aligned when synchronized, and misaligned when not synchronized. Right: Ypos used as a quantitative variable: the number of cycles is mapped to the distance between instructions (aaaaa: 2 cycles, bbbbb: 1 cycle).

Explore and leverage properties of visual variables. In a typical textual language, Ypos is used in an *ordered* but not *quantitative* manner. Since the distance between instructions has no meaning, a representation could vary distances to misalign statements and align synchronized statements only. In Fig. 20-left, the *selectivity* of the Ypos variable may help show at a glance the synchronization points and may remove false information conveyed by perfectly aligned statements. Distance can also be used to convey quantity. Fig. 20-right illustrates a representation that uses *Ypos as a quantitative variable* to depict two concurrent sequences of instructions. The number of cycles required by each instruction is mapped to the Ypos dimension. The larger the space after an instruction, the larger the number of cycles it takes to execute it. This gives a sense of the time spent on some parts of code, and may help balance the instructions in order to minimize wasted cycles while waiting for the concurrent process when synchronization is needed.

8. Threats To Validity

As stated previously, our goal is the production of knowledge about the phenomenon of code perception, but the explanations given in the paper are hypothetical and their validity may be questioned. We argue that the value of the paper lies elsewhere. For this work, we have used an abductive mode of knowledge inference [27]. Abduction consists in observing a phenomenon (b) then inferring hypothetical explanations (a and $a \rightarrow b$). For example, we have seen that the observation "Lisp parentheses are difficult to read" (b) may be explained by the fact that "parentheses are marks differing in shape only" (a) and "shape is not selective in this context" ($a \rightarrow b$ or *if marks differ in shape only, then marks are not selective*). Requalifying and explaining selected observations is the essence of abduction [27] and is a recognized scientific method of knowledge production. Compared to deduction and induction, abduction is the most fertile mode of inference to foster discovery [26](in 5.172): though it is essential to the advancement of knowledge, the elicitation of explanations (and thus hypotheses to be tested) is often overlooked.

If abductions are the most fertile mode of inference to foster discovery, they are also the least secure and as such need to have practical implications leading at least to mental tests. An explanation is worth testing if it has instinctive plausibility or reasoned objective probability [26](in 6.452). Since SoG has been used in other contexts successfully and has been experimentally assessed to some extent (see [38] for a review), we argue that its application to explain PL-related phenomena is plausible ([14] supports this claim).

The proposed framework relies on models, and as such is a simplification of reality. Even if the framework allows us to describe a number of the perceptual phenomena underlying the perception of code, some important phenomena may not have been identified because of the limited capability of the framework, or because their explanation or cause is different. Visual perception is complex and some visual operations may be bypassed because of specific conditions such as layout or the number of items involved. Furthermore, code representation is not the only factor that contributes to program understanding. Other cognitive factors, such as learning, expertise, API usability and documentation [13] contribute to our understanding of a program, and may influence the way the user perceives or scans the code.

The reader might disagree with the author's application and might be willing to offer alternative explanations using the concepts described here. Such discussions about explanation, qualification, performance prediction and task elicitation are enabled thanks to the framework. This is one of the results of this work: to provide language specialists with a commonly agreed set of concepts so that they may be able to agree and disagree on their respective analysis, or to agree on their disagreement.

The tasks that we elicited (finding a parenthesis, understanding the structure of the code, finding the location of states...) might be considered simplistic. However, they are related to the considered level of analysis i.e. the perception of code. At this level, we argue that they are realistic and important: higher-level cognitive tasks are influenced by lower-level ones as previously demonstrated [14].

9. Conclusion and Perspectives

Our contribution consists of new and plausible explanations about phenomena pertaining to the perception of code representation. We have captured a significant set of phenomena pertaining to code representation at the level of "the page of code" with ScanVis and SoG. For a framework to model and corroborate existing phenomena is a first level of validation. We also showed how the framework may enable one to compare the representations with respect to reading tasks. Furthermore, the generative aspect of the framework may enable language designers to find new ways of representing the code. The examples and the provided design principles may help explore the design space of code representation.

This work shows that code representation is not about aesthetics but performance, and should not be an art [4] but a science following principles of visual perception. To foster understanding of a program, a representation of code that follows those principles is not accessory, but mandatory. Therefore the account presented in this paper may extend the set of important aspects underlying programming languages: lexical (what concepts are), syntactical (how concepts articulate), semantic (what concepts mean), but also perceptual (how efficiently concepts are represented, with respect to programming tasks). This should be a concern for all computer scientists and programmers, be they academic or practitioner, as much as basic knowledge about programming such as "functional and imperative programming", or "static and dynamic typing".

The explanations we produced with an abductive mode of inference should also lend themselves to scientific tests [26]-6.452. Indeed, even if the present analysis is reasonable, it has not been assessed with user experiments in the context of programming languages. Some analyses in other contexts have been substantiated by controlled experiments [24, 25]. One of the works cited in the paper offers evidence of the impact of the spatial distance between marks on program understanding [14]. Thanks to the modeling of phenomena, and the process of task elicitation fostered by the application of the framework, we believe that our work can provide researchers with new material to better formulate the hypotheses regarding performance predictions, and to better design experimental methods to test those hypotheses.

Another perspective opened by this work is the unification of existing concepts. Unifying concepts has been a traditional goal in science (e.g., Maxwell's equations unifying electricity and magnetism, or the Curry-Howard correspondance between types and proofs) because this may lead to important discoveries and insights. Here the framework brings together many aspects of visual layout and appearance of programming languages and contradicts the traditional opposition between visual and textual languages. It also contradicts the usual wisdom that visual languages are by essence better than textual languages: most textual languages are displayed using positional variables and thus may use the perceptual system efficiently, while some so-called visual languages may use visual variables (icons (shapes), links) quite inefficiently. This should be of interest for any computer scientist, including software engineers who often use various UML diagrams (another visual language) to document their software.

Even if a complete and detailed design method is still missing, what the framework suggests is that such a method should use a "Programmer-Centered Design" approach: it should emphasize the act of designing representations targeted at tasks meaningful for end-programmers, and not designing the representation in isolation. Meanwhile, the fact that the framework may be used to compare representation should encourage programmers to expect justifications from language and IDE designers. They should be compelled to explain why and how the language designed is better than another with respect to objective criteria. This should diminish the risk of "religious wars"[1], since using a shared, consistent set of reference concepts would make the comparisons and justifications claims better supported.

Acknowledgments

We thank the members of ENAC LII and especially Stéphane Chatty for the numerous discussions we had on this work.

References

[1] Raymond, D. 1991. Reading source code. In *Proc. of the 1991 conference of the Centre for Advanced Studies on Collaborative research (CASCON '91)*. IBM Press 3-16.

[2] Abelson, H. and Sussman, G. 1996. *Structure and Interpretation of Computer Programs (2nd ed.)*. MIT Press.

[3] Moody, D. 2009. The 'Physics' of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Trans. Softw. Eng.* 35(6), 756-779.

[4] Green, R. and Ledgard, H. 2011. Coding guidelines: finding the art in the science. *Comm. ACM,* 54(12), 57-63.

[5] Conversy, S., Chatty, S., Hurter, C. Visual Scanning as a Reference Framework for Interactive Representation Design. In *Information Visualization*, 10, pages 196-211. Sage, 2011.

[6] Bertin, J. (1967) *Sémiologie Graphique - Les diagrammes - les réseaux - les cartes*. Gauthier-Villars et Mouton & Cie, Paris.

[7] Alan F. Blackwell and Thomas R.G. Green. (2003) Notational Systems - the Cognitive Dimensions of Notations framework. In *HCI Models, Theories, and Frameworks*, Morgan Kaufmann, pp. 103-134.

[8] P. J. Landin. 1966. The next 700 programming languages. *Commun. ACM* 9, 3 (March 1966), 157-166.

[9] McConnell, S. 2004. *Code Complete*, 2nd edition. Microsoft Press.

[10] Clifton, M. 1978. A technique for making structured programs more readable. *SIGPLAN Not.* 13, 4, 58-63.

[11] Crider, J. 1978. Structured formatting of Pascal programs. *SIGPLAN Not.* 13, 11 (November 1978), 15-22.

[12] Ramsdell, J. 1979. Prettyprinting structured programs with connector lines. *SIGPLAN Not.* 14, 9, 74-75

[13] T. Tenny. 1988. Program Readability: Procedures Versus Comments. *IEEE Trans. Softw. Eng.* 14, 9, 1271-1279.

[14] Miara, R., Musselman, J., Navarro, J. and Shneiderman, B. 1983. Program indentation and comprehensibility. *CACM* 26(11), 861-867.

[15] Bednarik, R. and Tukiainen, M. 2006. An eye-tracking methodology for characterizing program comprehension processes. In *Proc. of the 2006 symp. on Eye tracking research & applications (ETRA '06)*. ACM, 125-132.

[16] Sharif, B. and Maletic, J.I. 2010. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *Proc. of International Conference on Program Comprehension*, IEEE, 196-205.

[17] Feigenspan, J., Kästner, C., Apel, S., Liebig, J., Schulze, M., Dachselt, R., Papendieck, M., Leich T. and Saake, G. 2013. Do Background Colors Improve Program Comprehension in the #ifdef Hell? In *ESE 18(4)*, Springer, 699-745.

[18] Moher, T.G., Mak, D.C., Blumenthal, B., and Leventhal, L.M. 1993. Comparing the comprehensibility of textual and graphical programs: The case of Petri nets. In *Empir. Studies of Programmers: 5th Workshop*. Ablex, 137-161.

[19] T. R. G. Green & M. Petre (1992) When visual programs are harder to read than textual programs. *Proc. of the 6th European Conference on Cognitive Ergonomics (ECCE 6)*, pp. 167-180.

[20] Whitley, K. and Blackwell, A. Visual Programming in the Wild: A Survey of LabVIEW Programmers', *J. of Visual Languages & Computing*, 12(4), 2001, 435-472.

[21] Petre, M. 1995. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM* 38, 6 (June 1995), 33-44.

[22] Buse, R. and Weimer, W. 2008. A metric for software readability. In *Proc. of ISSTA '08*. ACM, 121-130.

[23] Treisman, A. (1982). Perceptual grouping and attention in visual search for features and for objects. *Journal of Experimental Psychology Human Perception and Performance*, 8(2), 194-214.

[24] Card, S.K., Mackinlay, J.D., Shneiderman, B., *Readings in Information Visualization: Using Vision to Think*. San Francisco, California: Morgan-Kaufmann, (1999).

[25] Cleveland, W., McGill, R., Graphical Perception and Graphical Methods for Analyzing Scientific Data. *Science*, New Series, 229(4716) (Aug. 30, 1985), pp. 828-833.

[26] Peirce, C. S. Collected Papers of Charles Sanders Peirce, edited by C. Hartshorne, P. Weiss, and A. Burks, 19311958, Cambridge MA: Harvard University Press.

[27] Douven, I. Abduction. *The Stanford Encyclopedia of Philosophy (Spring 2011 Edition)*, Edward N. Zalta (ed.)

[28] Dix, A. Theoretical analysis and theory creation. In *Research Methods for Human-Computer Interaction.* Cambridge University Press, pp.175195.

[29] Steele, G. and Gabriel, R. 1996. The evolution of Lisp. In *History of programming languages II*. ACM, 233-330.

[30] Whitley, K., Novick, L. and Fisher, D. 2006. Evidence in favor of visual representation for the dataflow paradigm: An experiment testing LabVIEW's comprehensibility. *Int. J. Hum.-Cmp. Stud.* 64(4), 281-303.

[31] Burnett, M. Visual Programming. In *Encyc. of Electrical and Electronics Engineering*, 275-283. Wiley, 1999.

[32] Conversy, S. Improving Usability of Interactive Graphics Specification and Implementation with Picking Views and Inverse Transformations. In *Proc. of VL/HCC*, pages 153-160. IEEE, 2011.

[33] Bragdon, A., Zeleznik, R., Reiss, S., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F. and LaViola, J. 2010. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proc. of CHI'10*, ACM, 2503-2512.

[34] Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B. and Kafai, Y. 2009. Scratch: programming for all. *CACM*, 52, 11, 60-67.

[35] Myers, B.A. 1991. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proc. of UIST'91*. ACM, New York, NY, USA, 211-220.

[36] Wiedenbeck, S. The use of icons and labels in an end user application program: an empirical study of learning and retention. *Behaviour & Information Technology*, 1999, (18)2, 68-82.

[37] C. Appert and M. Beaudouin-Lafon. (2008). SwingStates: Adding state machines to Java and the Swing toolkit. *Journal Software Practice and Experience*. 38, 11 (Sept), 1149-1182.

[38] MacEachren, A. M. How maps work: representation, visualization, and design. Guilford Press, 2004.